Purpose

The goal of this project was to simulate a cache using 128 different configurations. These configuration were generated using 4 variables: the cache size, the block size, the cache placement type, and the write policy. The cache sizes could be 1KB, 4KB, 64KB, and 128KB. The block size could be 8B, 16B, 32B, and 128B. The cache placement type could be direct mapped, two way, 4 way, and fully associative. Finally, the write policy could be either write back or write through. After running all these different configuration, I was to generate an output file specifying the configuration, the hit rate, the number of bytes written from cache to memory, and the number of byte written from the memory to the cache.

Implementation

I started by implementing the 1KB, DM, block size 8 cache. I didn't care about write policy in beginning. I made functions for building the cache, writing to, and reading from the cache, and tracked hits. Once this was working, I made a loop to go through the different cache sizes. Once this worked, I began working on implementing different cache placement policies.

I made sure to use a true LRU replacement policy. When writing to a 2W, 4W, or FA cache, the line at the specified tag will be chosen based on priority, with the least recently used line being chosen. We will check for hit/miss at this chosen line, and write to the chosen line

When reading from a line, we need to loop through the width of the line and check each tag and valid bit. If at least one of those is good, we'll return a hit. Otherwise, return a miss.

Once I implemented the cache placement policies, I added in the WB and WT method. For these, I wanted to track how many bytes were written from memory to the cache, as well as from the cache to the memory. For write through, this was simple. Every time we have a write operation, the cache is written a word (4 bytes) which then writes to memory.

On read, if we have a miss the memory writes a block to the cache. If we have a hit, no transfer from memory to cache or from cache to memory is done.

For write back, things were a bit more complicated. Every block now needed a dirty bit. If the dirty bit is set, it indicates that the cache and the memory are

inconsistent. Thus, every time we have a write we set the dirty bit, because we are only writing to the cache (memory will thus contain old data).

If we have a write miss, we must evict the old block. If the dirty bit is set, this means that memory and cache are inconsistent, thus we must write the block to memory. If we have a write hit, then no transfer takes place.

If we have a read hit, then no transfer takes place. We simply read what's stored in the cache. If we have a read miss, the location in the cache must be evicted. If the dirty bit is set, we must transfer the whole block to memory. Because what's in memory will then get passed to the cache, the two will become consistent, and thus the dirty bit will be set to 0.

This WB logic seemed solid to me, however in implementation I received incorrect results for my the number of bytes going from memory to the cache.

I then proceeded to generate the traces and plots asked for. In particular, I generated a trace to show the effects of associativity. The provided test1.trace did this quite well, because it had many addresses which wrote to the same index in the direct mapped case. By expanding the number of slots in this index, we were able to get a lot more hits when the same address was accessed later on. This is due to the fact that the address was not being overwritten.

From the associative.trace results, I created a plot for the case where the cache size was 1024 bytes, the block size 8 bytes, and the write policy was write through. This plot shows that as we move towards fully associative, our hit rate goes up. See **Figure 1.**

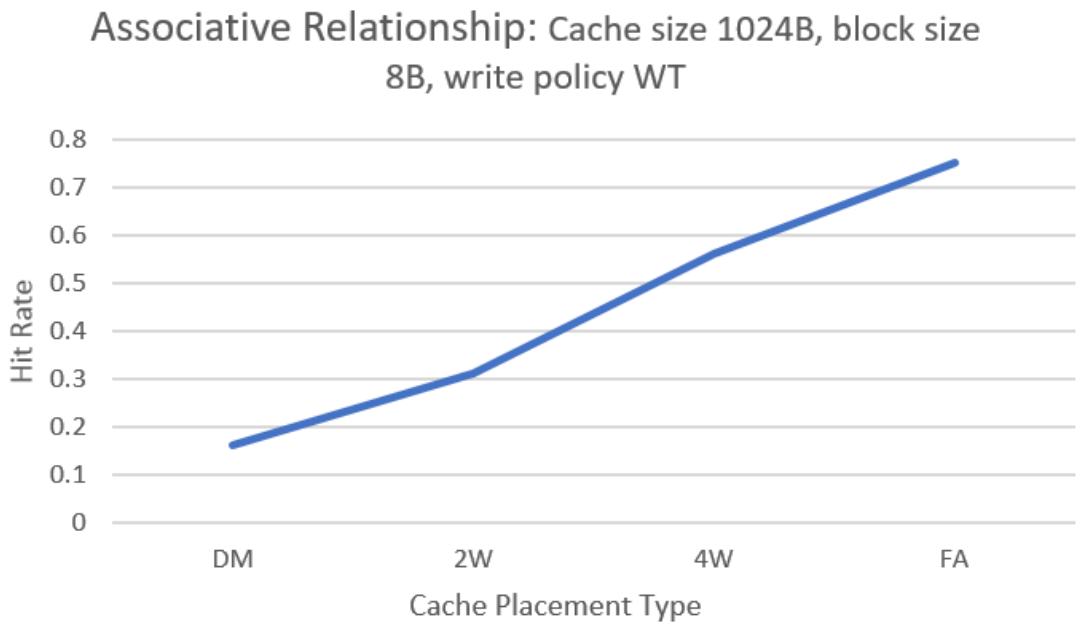

Figure 1. Demonstrating the improvement in hit rate as we move towards a fully associative cache

Next, I created a trace which demonstrated the effect of increasing block size. The provided test2.trace already did this quite well, so I chose it. After running test2.trace, I was able to produce a plot demonstrating the effect of increasing block size. Note that this plot was generated for a 1K, DM, WT cache.

As we can see, initially increasing the block size improves the hit rate. This is generally due to spatial locality, which can usually be seen in code and was written in manually in my trace in this case. As the block size continues to increase, the hit rate stabilizes and given large enough blocks would actually decrease. This is because we are

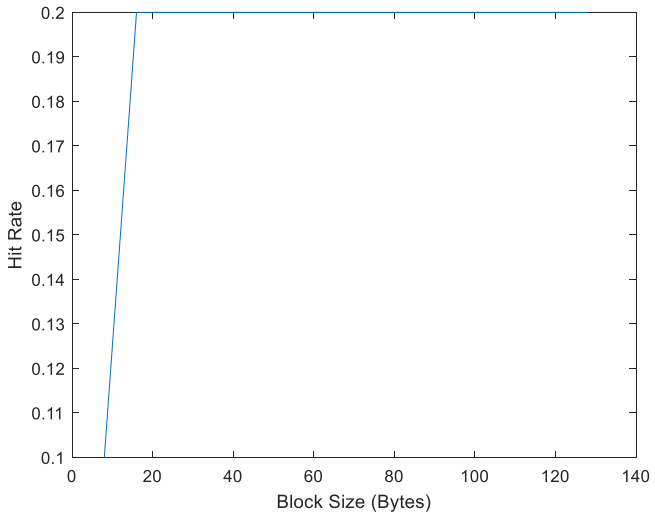

Figure 2. Block Size Effect on hit rate

decreasing the number of blocks available.